Advanced KIS Modeling

Kabira Infrastructure Server

Advanced KIS Modeling : Kabira Infrastructure Server

Published 27 Sep 2010

This document is intended to address *your* needs. If you have any suggestions for how it might be improved, or if you have problems with this or any other Kabira documentation, please send e-mail to pubs@kabira.com. Thank you! Feedback about this document should include the reference KISSY-32.2 . Updated on 27 Sep 2010 .

Contents

1

More KIS essentials

This chapter contains a collection of information that you need to successfully develop Kabira Infrastructure Server (KIS) applications. The chapter contains the following sections:

Prebuilt components and engines

This section talks about engines and libraries that provide various utilities and services, where to find out more about them, and how to use them.

Components

There are a number of useful components shipped with KIS. The reference to these is contained in the following on-line documentation:

- osstats component to get performance and other runtime statistics
- swbuiltin component for a range of utility interfaces described in the following paragraphs
- swmetadii component supporting meta-data and dynamic interface infocation
- sws component to support swalscript execution (now deprecated, applications should script the runtime using PHP instead)

Engines

KIS is shipped with several pre-built service engines:

- an administration engine `swadmineng`
- a web engine `osw`
- a distribution engine `swdisteng`

The administration engine `swadmineng` assists with engine upgrades and creates statistics reports.

The web engine enables access from the Internet to KIS engines with interfaces exposed through the PHP adapter.

The System Coordinator starts the administration and web engines automatically as part of system initialization in the Engine Control Center. Refer to Deploying and Managing ObjectSwitch Applications for more information.

The distribution engine `swdisteng` implements distribution services for all the other engines on the same node. More information about the distribution engine is presented in the registry section of Deploying and Managing ObjectSwitch Applications under Distribution.

Using KIS builtins

There are a number of utility functions built into KIS. These provide interfaces that your action language can use to:

- stop an engine
- perform various time and timing functions
- output a message to the trace log
- sleep for a specified interval
- purge an object from shared memory
- convert an object reference to a string representation and vice versa

You must import the appropriate component into any component you're building that uses these functions.

The following paragraphs describe some of these capabilities in general terms.

Using timers

The `swtimer` package contains a set of interfaces that provide timer functions. A timer lets you trigger some action after a specified period has elapsed.

Refer to the swtimer component documentation for more information about how to use timers:

$SW_HOME/userdoc/generic/components/swtimer/kis/

Time/date functions

KIS contains a `date` type in the `swbuiltin` package. You can declare a variable or attribute of type `date` :

```
attribute ::swbuiltin::date aDate;
```

and then you can convert between dates and longs using the interface functions also in swbuiltin. Refer to the on-line documentation for swbuiltin component for the time and time conversion operations available.

Model execution

The KIS Design Center produces an executable application engine from your model. The behavior of this application is controlled by the KIS runtime. Understanding the rules imposed by the runtime is crucial to:

- ensure that the application behaves as intended
- take full advantage of the features of KIS
- build applications which perform well
- avoid certain pitfalls

This section describes the model execution rules; the next section (the section called " Runtime implementation " on page 5) describes how the application server enforces those rules.

Starting, stopping, recovering

Operations can be designated to run when an engine is starting up, shutting down or recovering from an engine failure. The names of the properties used to specify this are `packageinitialize` , `initialize` , `terminate` , and `recovery` . Operations marked with these properties are referred to as boundary operations.

Boundary Operations When an engine is started boundary operations are executed according to the following rules:

- `already queued one-way and two-way operations and external (incoming from another engine) one-way and two-way operations are disabled.`
- `packageinitialize` operations are run first.
- external two-way operations are enabled.
- `recovery` operations are executed, if the engine is being recovered.
- `initialize` operations are executed.
- one-way operations (both external and internal) are enabled. This will include any operations that have been requeued due to recovery.

Boundary operations on singletons execute before boundary operations on non-singletons. Beyond that, the order of execution for multiple boundary operations of the same type is undefined.

`terminate` operations run when the engine is shutting down with external two-way operations enabled.

Event processing

KIS supports two types of events: synchronous and asynchronous.

Synchronous Events Synchronous events are used to implement two-way operations. They are always executed in the same transaction as the calling operation or state. The calling operation blocks until the two-way event is completed.

KIS optimizes the execution of most two-way operations when the implementation of the operation is available in the current engine. This optimization causes the two-way operation to be treated as a C++ function call instead of an event. This leads to higher performance by avoiding the overhead of an event and the copying of parameters in and out of shared memory.

Asynchronous Events Asynchronous events are used to implement one-way operations and signals. The runtime starts a new transaction and invokes the operation in the new transaction. If the runtime detects a transaction deadlock while running the operation, the runtime aborts the transaction, begins a new transaction, and restarts the operation.

Asynchronous events are queued to the target object and are executed one at a time, in the same order in which they were queued.

Only one asynchronous event can be processed by a particular object at a time.

Signals Signals are a type of asynchronous event used with state machines to cause state transitions.

Event reliability All events are guaranteed to be delivered. The only exception is that events to deleted objects are discarded.

Event queuing If an object is busy when an event is delivered to it, the event is saved for later consumption. Events are not lost.

Event processing An object processes the next event that is queued for it. There is no way for the model to examine the queue contents or to select which event to process next. When processed, a signal or one-way event causes one of three things to happen:

1. If the model specifies `ignore` for that event in the current state, the event is removed from the queue, with no further action.

2. If the model specifies `cannothappen` for that event in the current state, a runtime exception is thrown (for DEVELOPMENT builds), or else the event is simply discarded (for PRODUCTION builds) and a log entry is recorded. This `cannothappen` handling is the default event handling if neither (1) nor (3) are specified in the state model.

3. If the model specifies a state transition, then the object transitions to the new state and the action for that state executes.

After being processed, an event is discarded.

Two-way events are always processed.

Sequence of one-way event delivery—same sender An object receives events from a single sender in the same order that they are sent.

Sequence of one-way event delivery—different senders An object receives events from multiple senders in an indeterminate order. This order may or may not be the same order in which they were sent.

Event-to-self An event-to-self is processed before any other queued events to that object.

Sequence of two-way event delivery--different senders Two way events from multiple sources to the same object may execute in parallel. Transaction locking maintains data coherency. The order is not defined.

Action execution

Actions are used to implement operations and state actions.

Action concurrency among objects All objects are assumed to be executing concurrently. This means that you must construct the models so they would behave properly if all objects were in fact executing at the same time.

Action concurrency within an object One way operations and signals to a particular object are serialized. Two-way operations may execute in parallel with each other and with a one-way operation.

Action completion Once initiated, an action runs to completion. It may be interrupted, but it will never be discarded.

Action duration Actions take a finite, indeterminate amount of time to complete. You cannot make any assumptions about the relative time of different actions.

Concurrent data access

Because concurrently executing objects can access common data, the consistency of the data is at issue if no rules are imposed on it. (For example, you could branch when some attribute is true, only to have that value changed immediately after the branch was made, thus executing code that is inconsistent with the attribute value.) So KIS imposes the following rule:

Data consistency All the data accessed during an action remains unchanged, except by the action itself, during the execution of an action. This is done automatically by internal transaction locking.

Runtime implementation

A KIS engine includes a dispatcher that looks for events arriving on the event bus. Each engine has a known set of events that it accepts. Whenever a dispatcher detects an event, it handles it according to the event rules described in the section called “ Event processing ” on page 4. The dispatcher manages a pool of threads, so multiple events may be serviced at the same time. There are two basic types of events which cause work to be done in a KIS engine:

- asynchronous events (signals and one-way void operations)
- synchronous events (two-way operations)

Asynchronous event processing

An asynchronous event executes in its own transaction context; the event's delivery and execution is independent of the caller's transaction context.

Dispatch When a caller invokes an asynchronous event, the event is queued on the KIS event bus, but will not be dispatched until after the caller's transaction commits.

Handling Once an asynchronous event is dispatched, it is processed as follows:

1. the event is picked off the event bus by a dispatch thread
2. a new transaction is implicitly started
3. the dispatch thread executes the event on the target object
4. the transaction commits when the implementation (which may be an operation or a state action) completes

Synchronous operation processing

A synchronous operation executes in the transaction context of the operation or adapter that invoked the operation; no new transaction is created. The synchronous operation has no effect on the state of the caller's transaction.

Dispatch and handling When a caller invokes a synchronous operation that is not deployed in the same engine, the event is queued, dispatched, and executed in the caller's existing transaction as follows:

1. an event is placed on the event bus
2. the caller's execution is suspended
3. a dispatch thread executes the event on the target object
4. the dispatch thread notifies the caller's thread that it is complete
5. the caller's execution resumes

When a caller invokes a synchronous operation in the same engine, the event bus is omitted and the operation is called directly.

Object Lifecycle and Memory Management

The lifecyle of object instances in shared memory must be explicitly handled by the modeler using the `create` and `delete` action language verbs. Once an instance of an object has been created it will remain in shared memory until it is explicitly destroyed by the modler using `delete` . There is no automatic garbage collection of object instances in the runtime.

The runtime guarantees that memory associated with an object is always valid for the life of a transaction, even if the object is deleted in that transaction. Memory associated with destroyed objects is not deleted until the current transaction commits. This ensures that a modeler can never access an invalid memory location using action language and that aborted transactions correctly restore the state of the object.

Transaction processing

Transactions run implicitly throughout all KIS operations. Any change in KIS shared memory happens within the context of a transaction and is logged. In this way, any failure of a thread or an engine allows rollback to:

- restore entity state data to its pre-transaction state
- dequeue all events created by the transaction

Then the event that caused the transaction can be replayed. Important aspects of this transaction processing are:

- each event is guaranteed to be played once and only once
- the transactions work across distribution boundaries
- application data and state are preserved.

Normally, you don't need to control the transaction environment. However, if you explicitly spawn new threads, you must control transactions in the spawned threads explicitly. This is typically used for such things as "listeners" for protocol adapters.

Transaction rollback doesn't roll back processing done outside of the shared memory context. For example, if the action interacted with the physical world (e.g., directly turned a switch on or off), this would not be rolled back by the transaction rollback and replay. In fact, it would be set a second time.

Transaction Locks

In order to maintain consistent data across an entire transaction, the transaction places locks on objects accessed during the transaction.

With the exception of the `select using` clause, all of the locking mechanisms are transparent to you, but it is helpful to think about the types and amount of data being locked in a transaction.

Read and write locks There are read locks and write locks. Many readers may access an object simultaneously, but only one writer may access an object at a time. Read locks can be "promoted" to write locks in the course of a transaction. This occurs when a transaction holds a read lock on an object and requests a write lock. If more than one transaction concurrently attempts to promote the same read lock and automatic deadlock occurs in the other transaction(s). This is known as a promotion deadlock.

Entity locks Each object has a transaction locks that is used in the following manner:

Attaching an object does not take the lock:

```
for object in SomeObjectType
```

or:

```
object = otherObject;
```

Sending an event to an object does not lock the object:

```
object.someOperation();
```

Reading an attribute of an object takes a read lock:

```
if (object.someAttribute == true)
```

or:

```
someVariable = object.someAttribute;
```

This prevents any attributes of the object from being modified by any other transaction until this transaction completes.

Modifying an attribute of on an object takes a write lock:

```
object.someAttribute = true;
```

This prevents any attributes of the object from being read or modified by any other transaction until this transaction completes.

Modifying an attribute of an object when the read lock is already held causes a promotion lock:

```
if (object.someAttribute == true)  // read lock
{
     object.someAttribute = false; // promotion lock
}
```

Selecting an object takes a read lock on the object. Regardless of whether or not the where clause has complete key coverage.

```
select object from SomeObjectType where (object.name == "joe");
```

Selecting an object with the `using` clause allows explicit control over how the object will be locked. See the `select using` section in the Action Language Reference section.

Extent locks For object types that are `managedextent` there is transaction locking on the extent, the set of all instances of that entity type.

Creating or deleting an object will write lock the extent, preventing `create` , delete, iteration and `cardinality` for this object type in other transactions until this transaction completes.

Cardinality or iteration of the extent will cause it to be read locked, blocking other transactions from creating or deleting objects of this type until this transaction completes, but allowing concurrent iteration and `cardinality` in other transactions.

Extentless For object types that are not `managedextent` no extent transaction locks are taken for create, delete, cardinality and iteration. This allows for greater parallelism in a model, but also creates the possibility of inaccuracy in iteration and cardinality.

All object types are `extentless` by default, unless the `managedextent` property is used. `extentless` is still accepted as a property but is ignored.

Deadlocks

Since transactions are running simultaneously, it is possible to have deadlock situations. KIS automatically detects deadlocks and handles them in the following manner.

- one deadlock thread is chosen as a "winner" and allowed to complete

- the other deadlocked threads are “victims”. For each victim:

1.	the thread is killed
2.	the transaction is rolled back
3.	the transaction is restarted

These deadlock mechanisms are transparent to you. But deadlocks are expensive in a time critical environment, and you should try to avoid them.

In dispatch threads deadlocks are handled automatically. In spawned threads, you must catch the `ExceptionDeadlock` exception, then abort and rerun the current transaction.

Tip: avoiding deadlocks A strategy for deadlock avoidance is to look at potentially deadlocking transactions and try to access objects in the same order. For example, if One accesses entities in the order A, B, C, and thread Two accesses the same objects in a C, B, A order, deadlocks will be likely. If, on the other hand, both threads access the entities in the same order, there will be no deadlocks.

Tip: tracing deadlocks Transaction ids are useful in deciphering deadlock information. Each transaction is assigned a unique id. All trace messages report transaction ids in the format: `envId:tranId` . The environment id, `envId` , is unique within a node, and changes whenever a transaction environment is created. The transaction id, `tranId` , changes every time a transaction is begun.

Component Instantiation at Runtime

The Design Center creates components which are executed at runtime in a process called an engine. An engine can contain one or more components and the same component can be installed into multiple engines.

Access to a component at runtime is done through a stub. The stub provides access to the implementation of a component. The implementation is the compiled action language that is executed to provide the services of the component.

When a component is first initialized in the runtime it publishes all of it’s type information to the runtime. This type information describes all interfaces, entities, and other types that are implemented by this component. This type information is used by the runtime to create instances of objects, invoke operations, and access attribute data.

Once type information has been published to the runtime it remains available to the runtime even if the component is stopped or removed from the system.

When a new version of a component is installed, the new type information is compared to the existing type information associated with this component in the runtime. If the types can be upgraded automatically they are. If changes were made to the types that cannot be upgraded, the old types must be removed before the new component can be installed. The removal of the old types is handled in the Engine Control Center.

Once a component has published it’s type information it is possible to create object instances and access attribute data without the component running. This is because the runtime uses the published type information to provide these services.

The only exception is if the object was implemented using an adapter, e.g. persistence. In this case, the component must be running to access the object because this requires access to runtime adapter services provided in the implementation of the component.

Invoking an operation or trigger on an instance of an object implemented in a component always requires the component to be running on the node where the object was created.

2

Using C++ with KIS

This chapter shows you how to call C functions or C++ methods from Kabira Infrastructure Server (KIS) action language.

To call C functions or C++ methods, you must ensure:

- the code being called is thread safe; you can provide serialized access to unsafe methods via a singleton; however, if the code in question depends on signals or a non-threaded `errno` , it may not be called from KIS; be sure to read the man page to determine the MT compatibility of the function
- the code in question has been previously compiled and is available as a library
- the calling model includes special transaction and recovery handling for code that allocates resources

This chapter contains the following sections:

Simple Calls with no Resource Allocation

C functions or C++ methods which do not allocate resources can be called directly. For example:

```
// examples/generic/cpp_callout/sqrt.soc

package Math
{
#pragma include <math.h>

   entity SqrtImpl
   {
      double sqrt(in double param);
      action sqrt
```

```
      {`
         return ::sqrt(param);
      `};
   };
};
```

In this example, the C function `sqrt()` is being called from the math library. Note the "`::`" preceeding the call to `sqrt()`, which forces the global C function `sqrt()` function to be called. To reduce any chance of ambiguity, fully scope the name of the C function or C++ method call.

When building a component that calls external C or C++, you may need to specify the libraries implementing those functions. For example, the `sqrt()` manpage specifies that you must link the math library with `-lm`. The actual name of the library is "`m`". This must be listed as a `libraryPost` property in the component specification:

```
component CalloutExample
{
   // IDLos source file
   source sqrt.soc;

   // IDLos packages to be compiled
   package Math;

   // Extra library to link
   libraryPost=m;
};
```

Calls with Resource Allocation

Code which allocates memory or resources must be handled much more carefully. For example, consider the following package which intends to implement a reliable log-to-file mechanism:

```
// examples/generic/cpp_callout/log.soc

package Logger
{
   native FILE* FileHandle;

   entity LogImpl
   {
      attribute FileHandle   m_fileHandle;
         void open();
         oneway void log(in string message);
         oneway void close();
     };
};

action ::Logger::LogImpl::open
{‘
   FileHandle fileHandle =
      ::fopen(“mylog.log”,   "w+");
   self.fileHandle = fileHandle;
‘};

action ::Logger::LogImpl::log
{‘
   FileHandle fileHandle = self.m_fileHandle;
   ::fputs(static_cast<const char *>(message), fileHandle);
   ::fputs("\n", fileHandle);
‘}

action ::Logger::LogImpl::close
```

```
{‘
   FileHandle fileHandle = self.m_fileHandle;
   ::fclose(fileHandle);
‘};
```

Upon first inspection, this code may look fine. The reliable queuing mechanism in KIS guarantees that any `log()` message is eventually processed.

However, this code fails to handle two very important situations: aborted transactions and recovery.

Aborted transactions

Aborted transactions commonly occur in a deadlock situation, where a set of objects block each other while trying to access the same resource. You can code your actions to avoid deadlocks, but you have no control over the calling code. If it causes a deadlock, the entire transaction will be aborted, including the part that invoked your operation.

As a result, you have to assume that your actions might be called more than once for a particular event. If your action allocates resources, but doesn't handle this situation, you wind up allocating the same resource twice. Likewise, if your action frees a resource, it may be freed twice. Fortunately, KIS provides commit and abort triggers to help you identify these situations.

In the above example, aborted transactions are a problem in the `open()` , `log()` , and `close()` operations.

Recovery

Recovery happens when the engine process ends abnormally and then restarts. All events are re-played, but resources allocated to the process no longer exist. This is a problem for the `log()` operation, which now attempts to write to a closed file. KIS provides initialization hooks for startup that allow you to handle this situation.

The example below shows how triggers and initialization mechanisms may be employed to solve these problems.

Here now the IDLos for the example:

```
package Logger
{

   // Use native keyword to alias a C++ pointer type
   native FILE *FileHandle;

   //
   // Exception thrown if log file cannot be opened.
   //
   exception LogOpenException {};

   //
   // Public definition of Log object.
   //
   // The Log object uses one-way events to demonstrate
   // how KIS can queue work.  Calling log()
   // sends an event to the log object, but the caller
   // does not wait for the message to get written to the
   // file.
   //
   // Usage:
   //   Creation:  declare Log log;
```

```
//              create log values (fileName:"log.log");
//
//   Usage:     log.log("your message here");
//
//   Destruction:  log.close().  You do not need to
//              explicitly delete the log object.  It
//              will auto-delete once close() is called
//              after all pending log messages have been
//              written.
//
interface Log
{
   attribute string fileName;
   oneway void log(in string message);
   oneway void close();
};

//
// Private definition of Log object
//
entity LogImpl
{
   enum State
   {
      Uninitialized,
      Opened,
      Logging,
      Closed
   };

   //
   // Public API
   //
   attribute string       fileName;
   signal log(in string message);
   signal close();

   //
   // Private attributes
   //
   attribute State                m_state;
   attribute long         m_currentFilePosition;
   attribute long         m_previousFilePosition;
   attribute long         m_engineID;
   attribute FileHandle      m_fileHandle;

   //
   // Private lifecycle operations
   //
   void onCreate();
   trigger onCreate upon create;

   void onDelete();
   trigger onDelete upon delete;

   void onCommit();
   trigger onCommit upon commit;

   void onAbort();
   trigger onAbort upon abort;

   void openFile() raises (LogOpenException);

   // Stateset
   [finished={Close}]
   stateset
   {
```

```
         Open,
         Log,
         Close
      } = Open;

      transition Open to Log upon log;
      transition Open to Close upon close;
      transition Log to Log upon log;
      transition Log to Close upon close;

   };

   expose entity LogImpl with interface Log;

   //
   // A local entity with a package initialize operation
   // which will restore the file handles of any Log
   // objects in this engine.
   //
   [local]
   entity Recovery
   {
      [packageinitialize]
      void recover();
   };
};
```

Create trigger

The create trigger stores the engine ID for recovery (see the section called “ recover operation. ” on page 18).

```
action Logger::LogImpl::onCreate
{`
   if (self.fileName == "")
   {
      self.fileName = "example.log";
   }

   // Store our engine ID.  This will be required to
   // identify LogImpl instances that belong to this
   // engine during recovery.
    declare swbuiltin::EngineServices      es;
    self.m_engineID = es.getEngineInstance();

   // Initialize the file offsets and open the file.

   self.m_currentFilePosition = 0;
   self.m_previousFilePosition = 0;

   self.openFile();

`};
```

openFile operation

Since it is a two-way operation called from a create trigger, it is prone to deadlock from the caller. We will need to handle an abort so the file is not opened twice.

```
action Logger::LogImpl::openFile
{`
   // Take a write-lock on self.  We must do this to
   // guarantee we will not deadlock accessing our
```

```
   // own object when assigning to self.m_fileHandle

   self.m_state = Opened;

   // Enable the abort trigger.

   SWBuiltIn::enableAbortTrigger(self);


   // Allocate the resource

   char *mode;
   if (self.m_currentFilePosition > 0)
   {
      mode = "r+";
   }
   else
   {
      mode = "w";
   }
   declare FileHandle fileHandle =
      ::fopen(static_cast<const char *>(self.fileName),
      mode);

   // If open fails, we need to handle the error.  Throw
   // a user exception to the creator.

   if (fileHandle == NULL)
   {
      declare LogOpenException e;
      throw e;
   }

   // Reposition the file offset.  The first time this is called,
   // it has no effect since the offset will be zero.
   // Subsequent opens (such as after recovery)
   // will reposition the file offset to the last good location.

   int result = ::fseek(fileHandle,
         self.m_currentFilePosition, SEEK_SET);

   // Can't seek?  Close the file (and handle error...)

   if (result != 0)
   {
      // Can close file immediately because we are in the
      // same transaction
      ::fclose(fileHandle);
      self.m_state = Uninitialized;
      declare LogOpenException e;
      throw e;
   }

   SW_ASSERT( ::ftell(fileHandle) == self.m_currentFilePosition );

   self.m_fileHandle = fileHandle;

`};
```

Log operation

Since this is a one-way operation, it is not prone to deadlock other than the initial write-lock on self. As long as this write lock is taken before the external file is modified, no abort handling is required.

Were this a two-way operation, then you would need to be prepared to handle an abort and roll back the file pointer to the position at the start of the transaction. This handling is demostrated here as if it were necessary.

```
action Logger::LogImpl::Log
{`
   // Take a write lock on ourselves.  If a deadlock
   // happens here, we have not yet written to the file.

   self.m_state = Logging;

   // Need to store the file position BEFORE we update
   // the file, so the abort trigger know where to
   // seek to.

   self.m_previousFilePosition = self.m_currentFilePosition;

   // Enable the abort trigger

   SWBuiltIn::enableAbortTrigger(self);

   // Write to the log file

   int result = ::fputs(static_cast<const char *>(message),
            self.m_fileHandle);

   if (result > 0)
   {
      self.m_currentFilePosition += result;
      result = ::fputs("\n",self.m_fileHandle);
   }

   if (result > 0)
   {
      self.m_currentFilePosition += result;
   }
   else
   {
      // A write error has occurred.

      // In this example, we simply ignore the error and
      // throw away the log message.

      // You may wish to adopt a different strategy, such
      // as:
      //  o abort engine (call SW_BOMB())
      //  o open a new file
      //  o redirect output to stdout or stderr
      //  o call close(), which throws away existing log
      //      messages but no one can log new messages.
   }

   ::fflush(self.m_fileHandle);
`};
```

Abort trigger.

This is enabled when file is opened or written. We know which action to take based on the value of `m_state` . Note that the object state is the image prior to rollback, so all attributes assigned in the aborted transaction still have that data.

You cannot modify an object in an abort trigger, only examine its values.

```
action Logger::LogImpl::onAbort
{`
   declare FileHandle fileHandle = self.m_fileHandle;

   if (self.m_state == Opened)
   {
      // File was opened, we need to close
      ::fclose(fileHandle);
   }

   if (self.m_state == Logging)
   {
      // File was written, need to seek
      ::fseek(fileHandle,
         self.m_previousFilePosition, SEEK_SET);
   }
`};
```

Close operation

Since this is marked as a `[final]` event, the object is deleted. We leave the file closing up to the delete trigger (see the section called “ Delete trigger. ” on page 18).

```
action Logger::LogImpl::Close
{`
   // This is a final state, so object will be deleted.
   // Close is handed by the delete trigger.
`};
```

Delete trigger.

We use this to guarantee the file is closed, no matter how it is deleted. Note that even deleted objects can "reincarnate" due to deadlock, so a commit trigger is used to do the actual file close (see the section called “ Commit trigger. ” on page 18).

```
action Logger::LogImpl::onDelete
{`
   if (self.m_state != Uninitialized)
   {
      declare swbuiltin::ObjectServices os;
      os.enableCommitTrigger(self);
      self.m_state = Closed;
   }
`};
```

Commit trigger.

This is used to close the file so that it is not closed more than once.

```
action Logger::LogImpl::onCommit
{`
   declare FileHandle fileHandle = self.m_fileHandle;
   ::fclose(fileHandle);
`};
```

recover operation.

This operation is called whenever an engine is initialized, but before any events are fired. We need to reopen any files that were previously opened and restore the file positions.

```
action Logger::Recovery::recover
{`

   // Find only those LogImpl instances that
   // are implemented in this engine.

        declare LogImpl   log;
        declare swbuiltin::EngineServices     es;
        declare long    engineID es.getEngineInstance();

   // Recover each log instance by re-opening the log
   // and re-positioning the file offset

   for log in LogImpl where(log.m_engineID == engineID)
   {
      if (log.m_state == LogImpl::Closed)
      {
         // No need to close anymore
         log.m_state = LogImpl::Uninitialized;
      }
      else if (log.m_state == LogImpl::Opened
            log.m_state == LogImpl::Logging)
      {
         log.openFile();
      }
   }
`};
```

Careful use of triggers provides a means to manage external resources in the face of aborted transactions. Note that triggers like `abort` and `commit` do not allow you to manipulate the object itself. You may only read the object and perform external functions calls.

Likewise, a `packageinitialize` operation gives you the chance to initialize and restore any external resources before events are replayed.

You must give great care to the lifecycle of process resources. Generally, it is best to allocate and deallocate resources in create and delete triggers. If however, you must allocate a resource in an operation, consider making the operation a one-way so that it executes in its own transaction. This makes `abort` and `commit` trigger handling simpler.

Two-way operations are invoked immediately and thus can be reinvoked if the current transaction aborts. One-way operations are queued in the current transaction but are not invoked until the current transaction commits. The current transaction can abort and replay as many times as necessary, but you know the resource is not allocated or deallocated until the current transaction commits.

3

Developing for performance

This chapter explains how to improve the performance of Kabira Infrastructure Server (KIS) applications. Performance has been a primary consideration throughout the development of KIS: applications can exhibit very high performance together with all of the other benefits of KIS technology. But certain principles must be kept in mind as applications are developed.

The chapter describes generic performance considerations, as well as those particular to KIS. There are also some sections on understanding the runtime better in terms of performance issues, as well as a section on tools you can use to measure performance:

- the section called " Generic performance considerations " on page 21
- the section called " KIS-specific performance considerations " on page 23
- the section called " Tools for performance " on page 26

Generic performance considerations

There are many ways to improve the performance of an application that have nothing to do with the KIS architecture itself.

Avoid extra work

Nothing is always less expensive than something. Don't implement unnecessary functionality. In the requirement and design phases of a project, strive to make everything as simple and clean as possible. This pays tremendous benefits during the implementation, debugging, performance tuning, acceptance and support phases of a project.

Don't optimize prematurely

Do not begin optimizing an application before it works correctly.

Create tests for the application

Performance work usually involves optimizing or changing code paths. A full set of verification tests makes it possible to determine if an optimization has broken the correctness of the application. Even better is if these tests are automated.

Make the performance measurable

Don't optimize without measuring. Make the performance of your application easily measurable.

Find the hotspots

Do not optimize without properly analyzing where the application's performance is suffering. Use the tools and techniques described in the "Tools for performance" section of this chapter.

Avoid extra data

Insure that your data structures are appropriately sized. Most applications involve copying data from one place to another. Minimize the cost of this by minimizing the size of the data.

Use the simplest data structures that correctly describe your data. For example, don't a string to represent an octet or a character value.

Avoid unnecessary parameters.

Don't make the general case pay for special cases

Organize your model so that the overhead for handling exceptional conditions is only incurred when those conditions arise.

Avoid deadlocks

When locking multiple objects always lock them in the same order. Concurrently locking them in different orders can result in deadlock. Although many systems (including KIS) detect this condition, it is almost always less expensive to avoid it.

Avoid promotion deadlocks

When an object is going to be modified (written) within a transaction, take the write lock first, instead of the read lock. This avoids the possibility of promotion deadlock between multiple transactions. Again, although many systems (including KIS) detect this condition, it is nearly always significantly less expensive to avoid it.

Avoid resource contention

Avoid adding single points of contention to your application. If your application is to execute concurrent multiple paths, insure that the paths use separate resources.

Use indexes/keys

Use keys to navigate directly to objects instead of linearly scanning all objects.

Don't modify keys

The data that the makes up keys should remain mostly static during the life of the object. Set it up once at object creation time. Modifying keys causes significant expense in index management and also can reduce scalability by preventing concurrent keyed lookups.

Handle invariants at build time

If your application contains calculations that can be predetermined, do them once, either at compile time, or application initialization time.

Cache common resources

If the application repeatedly creates and destroys resources, consider managing a pool of these resources in order to avoid the overhead of creating and destroying them.

Avoid unnecessary context switching

As CPU speeds continue to increase at a rate faster than the speed of memory subsystems, the cost of context switching increases. When possible, avoid things that cause context switching such as resource contention.

Know when to break rules

Performance work, like engineering in general, involves making trade-offs. Some of these suggestions may contradict each other, and some may contradict good design practice. Be aware of the cost and benefits of performance optimizations.

KIS-specific performance considerations

The KIS architecture is designed for the development of high-performance applications. But it is possible to achieve poor performance. This section explains how to use the KIS architecture for best performance in your applications.

Only measure PRODUCTION mode builds

DEVELOPMENT mode builds have extra self-verification code compiled into them that can significantly reduce performance. This is useful during the development (or debugging) of an application, but is not appropriate for performance evaluation.

Partition the application into the fewest possible number of engines.

KIS applications run more efficiently in a single engine versus multiple engines.

The primary reason for this is that synchronous operation calls can avoid the Event Bus.

This avoids the overhead of copying the operation parameters into shared memory, sending an event to another engine, having that event serviced by another thread in the destination engine, having the destination engine copy the result data back into shared memory and send a response back to the orignal engine and thread.

If the operation can be found in the same engine, the Event Bus is completely avoided and the operation parameter data and results are directly passed via the generated C++ code.

A secondary reason is that operating systems are generally more efficient context switching between threads within a single process, than between multiple processes.

There may be circumstances where multiple engines are useful or even required:

- Fault isolation for fragile components.
- Component isolation for ease of debugging during development.
- Conflicting or special external library dependencies.
- System resource limitations (e.g. maximum file descriptors per process).
- Singleton component restrictions.

Always run with enough physical memory

KIS is designed to be a memory resident application. Insure that there is enough physical memory in the system to allow your application to run without swapping. At a minimum this means that the size of the KIS shared memory file must be smaller than that of physical memory. Also take into consideration the memory requirements of all of the processes on the system (including the application).

On Solaris the vmstat command line tool can be used to determine if swapping is occurring. See the man page for details.

Avoid excessive logging

By default KIS engine logging has warnings and fatal error messages enabled. These log messages occur infrequently, but are generally important. Much other logging information is available, often useful while developing and debugging applications. But logging has a performance price. Do not measure application performance with extra logging enabled.

Avoid iteration of extents

Use relationships to contain a group of non-unique objects of a single type, or a group of objects of multiple types. Use keys to navigate directly to unique objects.

Object attributes vs. relationships

Consider using an object attribute for a one to one relationship instead of a relationship. This uses less runtime overhead, but places a requirement on the model to explicitly check for the validity of the object attribute.

Accessors instead of operations

Although it goes against some good object oriented practices, consider replacing simple modeled accessors (operations that read or modify an object's attributes) with direct access to the attributes.

Pass objects instead of complex parameters

Parameters to operations must be copied in and out of the event bus. The more complex or lengthy the parameter, the more expensive the copy. The exception is passing objects. An object is passed by reference, which is a relatively small amount of data (16 bytes).

Don't use too many one-way operations or states

Excessive use of one-way operations can cause poor performance due to contention in the event bus, system memory allocator and dispatcher threads.

On the other hand, making the entire application one synchronous operation removes the ability to scale in parallel across multiple processors.

Flow control at one-way boundaries

One of KIS's great strengths is its ability to absorb work (in the form of one-way events) at burst rates much greater than the ability of the application to process them. But this strength comes at a price. Shared memory is consumed to queue these events, and some contention is caused when queueing the events.

Performance and shared memory footprint improvements may be seen if it is possible to flow control the work entering the system.

Use simple data types

KIS supports a rich set of data types. Simply put, the more complex the data type, the more expensive it is for the KIS runtime to manage that type.

- An octet or character is less expensive than a string.
- Bounded strings and arrays can be less expensive than unbound strings and unbounded sequences. But there are there is a cross-over point having to do with the number of elements and how they are modified.
- Un-nested data types are less expensive than nested data types.
- Fixed data types are less expensive than the "Any" data type.

Pre-allocate sequences

If you are going to be repeatedly appending to a sequence within one transaction, pre-grow the sequence to the desired size. This reallocates the sequence memory once, instead of once per element added.

```
self.sequence.length = 100;  // grows the sequence

for (elem = 0; elem = 100; elem++)
```

```
{
    self.sequenceAttribute[elem] = someValue;
}
```

Tools for performance

KIS provides several builtin features that can help in analyzing application performance. These include built-in statistics, logging facilities, and inexpensive support for timing from within a model.

Built-in Statistics

KIS can collect statistics about operations, transaction locks, operation call stacks and runtime mutex locks.

Because there are performance and memory costs associated with collecting these statistics they are disabled by default. Be aware of this effect when collecting statistics and measuring performance at the same time.

Operation statistics these statistics show how many times operations has been called, how many deadlocks have occurred within each operation and how many other exceptions have occurred. In addition, it shows the minimum, maximum, average and total execution times, in microseconds. Control is provided for enabling, disabling and clearing these statistics as well as generated a report of the currently collected statistics.

These statistics are the primary view into how a model is spending CPU cycles.

Transaction statistics these statistics show how many times each object instance has been read locked and write locked, how many times there was contention attempting to obtain the lock, how many deadlocks have occurred, how many promotion deadlocks have occurred, how many times the object has been destroyed while the lock was trying to be taken and the minimum, maximum, average and total time (in microseconds) spent trying to take the lock.

Call stacks On a per object type, per one-way operation (or state), per transaction basis, these statistics show the chain of operations called, what objects were locked at each level, how they were locked and any deadlocks encountered.

Mutex statistics This report shows all of the runtime internal mutex locks, how many times they have been write locked, read locked, how many times there was contention, and the minimum, maximum, average and total time (in microseconds) taken acquiring the mutex.

Other reports

KIS can also generate reports about current memory utilization, event queueing and which operation types are implemented by which engine.

Memory Utilization The first line of this report shows the current percentage of shared memory currently in use, and the total size of the shared memory file. The rest of the report shows distribution of allocations across the buckets in the allocator.

Queue Report This report shows, on a per queue (engine) basis, the number of events queued, and on a per channel (thread) basis, the events currently in service.

Subscription List This report shows on a per engine basis, which events are implemented in which engine.

Accessing statistics

Statistics can be accessed with a graphical interface via the Engine Control Center (ECC), via a modeled interface (osstats component) or directly from the command line with PHP.

via the ECC A graphical interface to the statistics is found opening the node, the system folder, selecting the administration engine and then clicking the Statistics button. This opens a window pane where different tabs and pull downs can be found for the various reports.

from a model A model can import the osstats component and directly access the statistics operations. See the osstats component on-line documentation.

from a script The osstats interface can also be invoked from a PHP. See the PHP section of Creating ObjectSwitch Applications for details on how to use PHP.

Log files and deadlock messages

Every KIS engine maintains a log file. By default this file contains warning and fatal error messages only. Via the ECC you can configure other logging options.

Messages of interest to performance work are generally the deadlock messages, which are warnings and thus enabled by default.

If the log files show deadlocks occurring more than infrequently, performance gains may be had by changing the application to avoid the deadlock conditions.

Deadlock messages come in several different formats and may change slightly from release to release, so this section presents the general style of these messages.

Trace messages contain a set of fields separated by the ‘ ’ character. The first several fields are fixed:

- `DateTime` when the deadlock occurred.
- `File LineNumber` is where the trace message was invoked. For deadlock messages this is not interesting as it refers to the runtime source file where the deadlock trace message is generated.
- `Thread ID` is identifies the thread that was running when the trace occurred.
- `Subsystem ID` identifies which area of the system the trace is from.
- `Severity` is either Debug, Info, Warning or Fatal.

The rest of the message varies with the particular message type. For deadlocks in will normally contain a transaction identifier and a description of the object attempting to be locked when the deadlock was detected. Some deadlock messages will also contain information about the type of deadlock and what operation was executing.

More information can be seen by enabling the ObjectServices Info tracing for the engines involved. This add extra information to the log containing all of the objects currently locked in transaction when the deadlock was detected, and how they were locked.

Another useful technique is to use the callstack statistics. See the section called “ Call stacks ” on page 26.

Measuring time intervals within a model

KIS provides a low overhead interface for collecting high resolution timestamps. It is available in the swbuiltin interface.

```
declare swbuiltin::TimeFunctions    timeFunc;
declare long long    startTime;
declare long long    finishTime;
declare long long    ticksPerSecond;
declare double    numberOfSeconds;

startTime = timeFunc.getTimestamp();

// invoke something here is to be be timed.

finishTime = timeFunc.getTimestamp();

ticksPerSecond = timeFunc.getTimestampResolution();
numberOfSeconds = (finishTime - startTime);
numberOfSeconds /= ticksPerSecond;

printf(“%.6f seconds\n”, numberOfSeconds);
```

Non KIS Tools

On a Solaris system the vmstat command is very useful for determining processor utilization, paging and swap activity and context switch rates.

top is another utility similar to vmstat. It may not be available on all systems.

Another set of interesting utilities on Solaris are the /proc utilities located in /usr/proc/bin. pstack is generally useful for debugging, while pmap may be of use to those trying to understand the low level details of process address space. See the man pages for details.

4

Building distributed applications

This chapter discusses how to build and deploy distributed applications with Kabira Infrastructure Server (KIS). It contains the following sections:

the section called " Overview " on page 29 — This section provides an overview of the KIS distribution architecture.

the section called " Distribution concepts " on page 30 — This section introduces distributed concepts and how they are implemented in KIS.

the section called " Using distribution from a model " on page 36 — These section discusses how to access distribution from a model and the impact it has on normal modeling concepts.

the section called " Administration " on page 39 — This section discusses the configuration and administration of distributed applications. It presents everything needed to deploy and monitor a distributed application.

the section called " Examples " on page 44 — This section presents some example distributed applications that show how you might use the distribution functionality in KIS to solve real world problems.

Overview

The unit of distribution in KIS is an object. Objects are distributed across KIS nodes. A node is a shared memory file, not a physical machine. This allows a single machine to support multiple KIS nodes.

Access to a distributed object is through the object reference. Every object reference includes data to identify the node where that object was created. You control the node where an object is created by design and management techniques such as:

- the use of factories

- naming services
- default behavior that you define for the object type when it is deployed

The same instance of an object cannot exist on two nodes. Copies of an object's state may be located on multiple nodes to improve performance or robustness, but the master copy is located on a single node.

An object's behavior executes on the node where the object was created. Any operations invoked on an object are sent to the node where the object was originally created and executed there. This includes all user defined triggers for an object.

Objects of the same type can exist on multiple nodes. You can achieve this by installing components containing the same types on multiple nodes. This is necessary to support data partitioning and caching schemes.

Distribution concepts

This section presents important distribution concepts and explains what they mean in the KIS runtime.

Location transparency

KIS provides location transparency for objects. This means that when your application accesses an object, its location is transparent—it may be local or on a remote node.

Location transparency is accomplished through the use of distributed references. All objects created in the runtime have a distributed reference that contains the location where the object was created. Operations invoked on an object are routed back to the location where the object was created.and executed on that node.

Attributes are accessed on a local copy of the attribute data in memory. See the section called " Caching " on page 33 for details on how object data is cached.

Location transparency is also provided for relationships and extents. Relationships and extents can contain object references that are located on both the local and a remote node. When a relationship or extent is iterated in action language, the runtime transparently provides access to any remote objects in the relationship.

Locations

Every KIS node in a distributed environment has both a location code and a location name. Location codes and names must be unique across all nodes that will be communicating in a distributed environment. The default values for these are:

Location identfier	Default value
Location code	The time in ticks since the host system was started
Location name	The local host name

Both of these defaults can be changed to allow multiple KIS nodes to run on the same host or to support a multi-homed host.

A location code is a numeric identifier that is encoded in every object reference associated with objects in the KIS runtime. This location code is used by the runtime and the distribution engine to determine the actual network location of the object.

A location name is a human-readable string associated with every KIS node. The location name is used to configure directed creates.

Discovery Services

Discovery services provide support for runtime discovery of location information.

Locations The location discovery service provides runtime mapping between a location code or location name and an actual network address. This mapping is done at runtime so network specific addresses don't need to be encoded in object references.

The location discovery service performs location discovery in two ways:

- static discovery using configuration information
- dynamic discovery using a UDP broadcast protocol

The system administrator can optionally configure the mapping between a location name and a location code/network address using the administrative GUI. This is only required if UDP broadcast cannot be used for location discovery. An example of when this would be necessary is if the remote node is across sub-net boundaries where broadcasts are not allowed.

If configuration information is not provided for a location name, UDP broadcast is used to perform dynamic location information discovery. This has the advantage that no configuration for remote nodes has to be done on the local node - it is all discovered at runtime.

Location discovery is performed in the following cases:

- A directed create to a remote node
- A dispatch on a remote object

When an object type is configured for directed create, the location on which the create should occur is specified using a location name. When a create of this type is done, a location discovery request is done by location name, to locate the network information associated with a location name if the network information is not already known on the local node.

When an operation is dispatched on a remote object, a location discovery request is done by location code, to locate the network information associated with a location code, if the network information is not already known on the local node.

Types

Type information for all objects installed on a node is broadcast to all other nodes when the distribution engine starts up. As new types are added to the local node they will be broadcast to other nodes in the distributed network.

When type information is received on a node, a local copy of the type information will be created if it does not already exist. If that type is already installed on that node, or the type information was received from another node, the type just received matches the type on the local node.

Type conflict If a new type does not match the type on the local node, information about the mismatch is stored in a type-mismatch table. Whenever data is marshalled up for this type, either from the local node, or when it is received from a remote node, the type mismatch table is checked to see if the type matches. If the location/type combination is found in the type mismatch table, a type conflict system exception will be raised by the runtime and returned to the originator of the request.

The type conflict system exception is defined in the swbuiltin component. It is defined as:

```
swbuiltin::ExceptionTypeMismatch
```

Distributed Transactions

KIS provides support for two styles of transactions:

- asynchronous
- synchronous

Asynchronous transactions are used for oneway operations and signals to a state machine. KIS executes this operation on the remote node in a different transaction from the one in which it was sent. The transactional guarantees ensure that this operation is executed once and only once on the remote node.

Synchronous transactions are used for two-way operations. KIS creates a global transaction and uses this global transaction to execute the two-way operation on the remote node. Multiple two-way operations executed from the same action will all be part of the global transaction. This includes two-way operations that are started on the remote node. All of these two-way operations will be committed or aborted as an atomic unit of work.

Locking Asynchronous transactions do not perform any distributed locking.

Synchronous transactions take locks on distributed objects. These locks are held until the transaction is commited or aborted.

Deadlock detection Distributed deadlock detection is required only for synchronous transactions. Asynchronous transactions do not take any distributed locks so it is not possible for a deadlock to occur.

Deadlock detection for a distributed transaction is done using a configurable time-out value. If a lock cannot be obtained on a remote node, on behalf of a distributed transaction, within the configured time-out period the caller is notified and the transaction is aborted, releasing all locks, and it is restarted.

Because distributed deadlock detection is based on a timer, models that have distributed deadlocks will perform very poorly. The only way to avoid this is to avoid all deadlocks in a distributed application.

Abandoned transactions Transactions can only be commited or aborted by the initiator of the transaction. This means that any global transaction executing on a remote node cannot commit or abort until the node initating the transaction explicitly indicates that this should happen.

In normal operation this works fine. However, in the case where a node that initiated global transactions fails, the transaction will remain pending on all remote nodes until the initiating node is restarted. If the initiating node never restarts, then the transaction is abandoned.

The distribution management console can be used to commit or abort abandoned transactions.

Caching

KIS provides support for two distinct types of object data caching: passive (or "pull") caching, and group broadcast (or "push") caching. The difference between these is:

passive	Passive caching copies attribute data to a remote node only when an object instance is accessed using an object reference.
group broadcast	Group broadcast caching automatically propogates allobject creations and deletions and updates to attribute data to all remote nodes configured to receive the updates using a cache group.

Once data is cached on a remote node, the data is refreshed based on the cache polices described below. All attribute access is done using the local cached copy of data. This avoids the network round-trip required by other distribution technologies to access object attribute data.

In all cases, modifications to an object's attribute data on a remote node are written back to the node on which the object was originally created.

Cache policies Every distributed type has a cache policy. The cache policy controls when cached data is considered stale and should be read from the node on which the object was created.

The following cache policies can be configured for a type. These cache policies affect the behavior of an object that is being accessed on a remote node. The local node for an object is the one on which it was originally created.

cache policy	meaning
always	The cached copy is always considered valid. It is never refreshed.
never	The cached copy is always considered stale. Every access to this object will cause the cached data to be refreshed.
once	The first time a reference is accessed, the cache is considered stale. After that the cached copy is always considered valid. It is never refreshed again.
timestamp	The cached copy is considered valid for a configured amout of time. The amount of time after which it is considered stale is controlled by a cache time. If the object is accessed after cache time has expired since it was originally read onto the node, it will be refreshed.

Types that are configured for group broadcast caching should be configured to have a cache policy of always. This is because any updates made to instances of this type will be pushed out to all nodes in the broadcast group keeping them in sync. If the cache policy is not always, a node may refresh the data again when it is accessed on the local node. The Distribution Administration User Interface will display a warning if this is attempted.

Extent Caching Extents have a cache policy of always. When an extent is accessed, only object references on the local node are returned. References get into the local extent either because the

object was created on the local node, or it was pushed to the local node because group broadcast caching is enabled for the type.

Relationship Caching Relationships use the cache policy of the "from" object in the relationship. The relationship is refreshed from the node where the "from" object was created only when the cache policy of the "from" object indicates that it is stale.

For example, if the "from" object in a relationship has a cache policy of always, the relationship will never be refreshed from the node where the object was created.

Flushing the Cache Distributed objects can be manually flushed by a modeler from a remote node. When an object is explicitly flushed by a modeler, the cache policies are ignored.

A type can be configured for the following flush types:

- none - this type cannot be flushed
- data - only an object's attribute data is flushed from memory. Object references are maintained for this object in it's extent and any relationships to this object.
- all - an object's attribute data and all references are flushed from memory. This implies that an implicit unrelate is done for this object from any other objects to which it is related when it is flushed from memory.

Objects that have only their data flushed from memory can still be navigated to using their extent and also via any relationships from other objects to which they are related. If an objects data and references are flushed from memory, this object can no longer be navigated to on the node from which it was just flushed.

The swbuiltin component provides the primitive for flushing objects from memory. To use this primitive the swbuiltin component must be imported into the distributed model. Here is a snippet that shows how an object can be flushed from memory.

```
declare swbuiltin::ObjectServices   os;

if (os.flushObject(myObject) == true)
{
   // all is well
}
else
{
   // flush failed
}
```

Cache groups Cache groups provide support for pushing cache data to a configured cache group by mapping the cache group to a set of network nodes. Cache groups are used when group broadcast caching is configured for a type.

Nodes are mapped to a cache group by the runtime by examing all types on the local node and determing all of the cache groups configured for these types. This is the list of cache groups that this node participates in.

Asynchronous vs synchronous Creates, writes, and deletes can be configured to occur either asynchronous or synchronously with respect to any remote nodes.

If these operations are defined to occur asynchronously they will occur in a separate transaction on the remote node than they did on the local node. This implies that there is data inconsistency between the two nodes for a period of time.

If these operations are defined to occur synchronously they will occur in the same transaction on the remote node as they did on the local node. This implies that there is always data consistency between two remote nodes.

Asynchronous operations improve the overall performance of a distributed system because no remote locks are held. They also avoid the overhead associated with a distributed transaction. The downside is that there can be data inconsistency in a distributed system at a given point in time.

Asynchronous creates Asynchronous creates cause an object to be created in a separate transaction on a remote node. Because the create is done in a separate transaction on the remote node there is no way for the runtime to report a duplicate key error back to the modeler. If a duplicate key is detected on the remote node, the create is not performed and a warning message is logged.

Reading and writing data Object attribute data is transparently read from and written to a remote node when attribute data is accessed on a local node.

Read operations are dispatched to a remote node to read attribute data depending on whether the cached data on the local node is stale or not. If the local cache is stale, a read will be done when an attribute is accessed. It will complete before the get of the attribute data returns to the caller. All reads are done on the remote node in the same transaction in which the attribute access occurs - in other words, they always execute synchronously.

When an attribute associated with a remote object is modified on a local node, a write is dispatched to the remote node to update the attribute data on that node. This write can occur in the same, or a different transaction depending on whether writes are configured to execute asynchronously or synchronously for the type.

If writes are configured to be performed asynchronously for a type it is possible that that target object of the write on the remote node has been deleted. This error is detected by the runtime and the write is discarded. A warning message is written to the trace file.

State conflicts A state conflict is reported by the runtime when a write operation from a remote node detects that the data on the local node has changed underneath it. This is possible in a distributed system because the object may be modified from multiple nodes in the system.

State conflicts are handled differently depending on whether writes are configured to be executed asynchronously or synchronously.

When writes are configured to execute asynchronously, the state conflict is not detected until the write is executed on the remote node. This is in a different transaction than the one the modified the object data. If a state conflict is detected, the data is discarded and a state conflict trigger invoked for that object (if one was provided by the modeler) on the remote node.

When writes are configured to execute synchronously state conflicts are handled transparently. If a state conflict is detected on the remote node an error is returned to the local node where the cache is flushed, the transaction will be rolled back and replayed. The modeler is never aware that a state conflict occured.

Using distribution from a model

This section provides information on how to use distribution from a model.

Accessing remote objects

Initial references

Directed creates

Factories

Attributes

Object state data is always cached on the local node. This implies that all attribute access on both local and remote objects are done at memory speeds.

An attribute get returns the current value for that attribute in the local cache. When the cache is refreshed is controlled by the cache policy for the type.

An attribute set modifies that current value for that attribute in the local cache. If this is an instance that was created on the local node, the modification is only propagated out to other nodes if this type has been configured for group broadcast caching. If this is the case, the update is sent to all of the nodes configured in the cache group.

If this instance is a remote object, a write is dispatched to the node on which the object was created when this object handle goes out of scope. A state conflict may be detected if the object has been changed on the remote node since the local cache was last updated.

In general, it is better to not allow attributes to be set through an interface. Attributes should all be defined as readonly to prevent this. This ensures that all updates are done using operations or signals, minimizing state conflicts.

Operations and Signals

All operations and signals for an object are executed on the node where an instance was created. The KIS runtime uses the object reference to determine the node on which the instance was created. If it is a remote object, the runtime marshalls up the operation or single and transparently sends it to the remote node where it is executed.

The fact that operations and signals are executed on the node where an instance was created is an important feature that modelers take advantage of to reduce the potential of state conflicts. By only allowing object state to be updated from an operation or signal, this forces all updates to be done on a single node. If updates are allowed using attribute accessors, the updates are being done on any node where the instance is cached. This increases the likelyhood of state conflict because there is no synchronization between all of these updates.

Triggers

Like operations and signals all triggers associated with a type execute on the node where an instance was created. This includes the relationship relate/unrelate triggers. The KIS runtime uses the object

reference to determine the node on which the instance was created. If it is a remote object, the runtime marshalls up the trigger and transparently sends it to the remote node where it is executed.

Commit and Abort Object types that have commit or abort triggers defined cannot be configured for distribution. When the distribution component is started it will report warnings in the swdisteng.log for all types that contain commit or abort triggers that cannot be configured for distribution. Here is an example of the warning:

```
002-02-17 09:52:00 service.cpp (391 ) 1 D   WARN  Detected commit/abort trigger for type
 remote::E, unable to configure this type for distribution
```

State Conflict The state conflict trigger is called by the runtime when an asynchronous write from a remote node detects that an object's data has changed on the remote node when compared with the cached copy on the local node.

A state conflict trigger is invoked on the target node of the asynchronous write. As mentioned above, state conflicts detected as part of synchronous writes on remote nodes are handled transparently by the runtime.

A state conflict trigger must be a void operation with no parameters. It can be a two-way, signal, or oneway operation. Here is an example of declaring a state conflict trigger:

```
entity E
{
    void stateConflict();
    trigger stateConflict upon state-conflict;
};
```

When a state conflict trigger is executed it has access to the current object attribute data, not the version of the attribute data that was in the cache on the remote node which sent the asynchronous write. The update from the remote node is discarded.

Relationships

Roles in a relationship can contain both local and remote object references.

When a relate or unrelate is done, the changes to the roles are propagated to the nodes where the related objects were created. This allows distributed relationship maintenance to occur.

Relationship information associated with a remote object reference is refreshed based on the cache policy associated with the "from" object type. Relationship changes are not propagated to remote nodes as part of group broadcast caching.

The implications of this are that it is possible to have stale relationship information a node if the cache policy of the object type indicates that the cache data should not be refreshed and the relationship has changed on a remote node.

Singletons

Distributed singletons only work if directed create is enabled for the singleton type. If directed create is not configured, multiple instances of the singleton can be created if a create singleton is done on different nodes.

Attempting to enable push caching for a singleton type will result in a runtime error when distribution is started on a node.

Using a distributed reference to a singleton which already exists on the local node will result in an ExceptionObjectNotUnique exception.

Extents

Global extents are only maintained if an object type is configured for push. As object instances are pushed out to all nodes, the extent on the node is updated to contain references to all instances in the distributed system.

If an object type is not configured for push caching, the extent on a node only contains the instances that have been created on that node, or pulled to the node in some other way (remote operation invocation, name service, factory, etc.).

Keys and Indexes

Like extents, global indexes are maintained only if an object type is configured for push. As object instances are pushed out to all nodes, the index on the node is updated to contain the key data for all instances in the distributed system.

If an object type is not configured for push caching, the index on a node only contains the key data for objects that have been created on that node, or pulled to the node in some other way (remote operation invocation, name service, factory, etc.).

Configuring a node for distributed operations after it has keyed objects created on it can cause problems. The reason for this is that as objects are pulled or pushed onto the local node they may have duplicate key values. These will be reported as duplicate key exceptions on the local node.

For distributed types, the action language below may potentially throw a duplicate key system exception because there is no distributed key manager.

```
select obj from MyObject where (obj.aKey == 1);
if (empty obj)
{
    create obj values ...
 }
```

You must handle the duplicate key exception by reworking this action language to:

```
declare boolean done = false;
declare long    keyValue = 1;
declare E       e;

while  (done == false)
{
    try
    {
        select e from E where (e.m_key == keyValue);
        if (empty e)
        {
            create e values (m_key:keyValue);
            done = true;
        }
    }
    catch (swbuiltin::ExceptionObjectNotUnique)
    {
        //
        //      Generate a unique key
        //
        keyValue +=1;
```

```
    }
}
```

Guidelines

Here are some general guidelines that should be followed by developers using distribution.

- All modifications to an object should be done on one node. This reduces the chance of state conflicts which cause performance degradation. The best way to enforce this is to make all attributes visible in an interface readonly and use operations to perform the updates.
- Use async operations and state machines. These avoid the transaction overhead associated with synchronous operations which require global distributed transactions.
- Eliminate distributed deadlocks from an application. Distributed deadlock detection uses a timeout to detect a deadlock. This implies that a distributed transaction will wait the entire value of the timeout value before a deadlock is reported. During this period of time this transaction is stalled.
- Factories or directed create should be used to create an object instance on a specific node.
- Evaluate which nodes application components must be installed on. Types configured for group broadcast caching require that the application component be installed on all nodes that participate in the cache group. Applications that use directed creates and factories do not require the application component to be installed on all nodes in the distributed network.

Administration

This section describes how to start the distribution services on an KIS node. It also provides information on how to configure and monitor an KIS application for distribution.

Starting Distribution Services

By default a KIS node does not have distribution services enabled. There are two components shipped that are used to enable distribution services on a node. They are:

- `swdisteng.kab` - Runtime distribution services
- `swdistadmin.kab` - Administrative services for distribution

The swdisteng.kab component provides all of the runtime services required to support distribution. Adding this component to a KIS node makes that node visible in a distributed environment.

The swdistadmin.kab component provides the administrative services required for distribution. This includes configuration and monitoring services.

The distribution components are added to a node like any user built component - using the Engine Control Center (ECC). The distribution components are located in:

```
$SW_HOME/distrib/<platform>/component
```

To enable a node for distribution use the ECC “Add Component” dialog and check the swdistadmin.kab component and then “Ok”. This will load both the Distribution Adminstration and the Distribution runtime services. The node is now enabled for distribution.

Distribution Administration Tool To access the Distribution Administration tool, click on the Distribution Administration component. A "Distribution Adminstration" button will appear in the right panel. Click this button and the Distribution Adminstration tool will be displayed.

The Distribution Engine and Distribution Administration Engine must be running before you can use the Distribution Administration tool. If these engines are not running the Engine Control Center will display a message indicating that these engines must be started.

The configuration and monitoring information associated with distribution is accessed using the tabs at the top of the Distribution Administration tool.

Configuring Distribution

All distribution configuration is done at deployment time not at build time. Most distribution configuration is done using the Distribution Administration tool.

On multi-homed hosts, you can select which network interface to use for the distributed discovery protocol. You can configure this in a deployment specification. Refer to "Deploying and Managing KIS Applications" for more details.

Global Settings The Global Settings tab contains configuration information that is global to all of distribution.

Types Configuration information for a specific type is configured under this tab. The only IDLos type that can be configured for distributed behavior is an entity. Attempting to configure any other IDLos type for distributed behavior is an error, this includes interfaces. There is no error reported by KIS for this error. The configuration information for the type is just ignored.

When the Distribution Administration tool is started it gathers information on all types currently deployed on the node. These types are available for configuration automatically. It is also possible to configure a type using the scoped name of the entity before a component implementing that type is deployed on the node. This is done using the "Add Type" button.

The type configuration information for distribution is read by the distribution component when it is started. This implies that any configuration changes made to a type while the distribution component is running are not reflected until the distribution component is restarted. No other components on the node must be restarted for them to take affect.

Type configuration information is specific to the objects implemented on the node. If a type is deployed on multiple nodes, the configuration values for that type should be consistent. This is because type discovery uses the first type seen. If the types are configured differently, the distributed network would have inconsistent type configurations.

Distribution and Persistence A type can have only a single object cache type. The object cache type configured in the Perrsistence Adapter takes precedence over the one configured in distribution. The distribution component will skip configuring the distribution cache type if it detects a persistence object. A warning will be written to the log file.

Discovery This tab contains the configuration values for the static discovery mechanism.

Transactions This tab contains configuration information for global transactions. It also contains monitoring information for active distributed transactions. The monitor facilities are described below.

Type Conflicts

Monitoring Transactions

Distributed transactions can be monitored from the Transactions tab. Any active distributed transactions are displayed. A distributed transaction is one that spans nodes.

Commiting and Aborting Abandonded Transactions

Monitoring Requests

A request is an outstanding one-way operation that has not received a response back from a remote node. Requests are used to ensure once and only-once delivery of work to a remote node. They are different from a transaction because they handle the non-deterministic failure conditions when using a network connection between two nodes.

The Distribution Administration Tool provides support for monitoring these outstanding requests from the "Requests" tab. In a normal running system it is very unlikely that any request information will be displayed on this screen since requests are very short lived.

Request information can be displayed in this report for long periods of time in the following cases:

- Remote distribution component has a mis-configured thread pool which has too few threads allocated for processing work from remote nodes. This is a transitory condition that will be automatically resolved by the runtime by creating more threads.
- A bug in the runtime. If requests are left abandoned in this report for long periods of time please report it to support.

Configuration Reference

This is a summary of the configuration parameters for distribution. These parameters are all configured using the Distribution Adminstration Tool.

Global Settings

The configuration information that is global to distribution services. This configuration information is accessed under the Global Settings tab.

Distribution configuration The global information in the following table applies to distribution generally.

Value	Default	Description
Node Hash Size	101	Hash table size of all open node connections.
Type Hash Size	101	Hash table size of local types pushed to remote nodes.
Request Hash Size	101	Hash table size for pending one-way requests to remote nodes.
Group Hash Size	101	Hash table size for groups associated with each remote node.

Dynamic discovery configuration The global information in the following table configures how dynamic discovery works.

Value	Default	Description
Location Code	Variable	Assigned by distribution services. It cannot be changed.
Broadcast Port	8866	UDP listen port number for location mapper. The discovery protocol is sent and received over this UDP port.
Location Hash Size	101	Size of the hash table for the location information cache.
Resolve Retry Count	0	The number of times a location mapping broadcast should be retried before a service unavailable exception is raised. A value of 0 indicates retry forever.
Resolve Delay Seconds	1	The number of seconds between location code mapping broadcast retries.
Conflict Delay Seconds	1	Number of seconds the location mapper will wait to determine an existing node has this location code taken already.

Transport configuration The global information in the following table configures the distribution transport protocol.

Value	Default	Description
TCP Port	5557	The listener port for inbound connections.
Retry Count	0	The number of times a connection should be retried before a resource unavailable exception is raised to the application. A value of 0 indicates retry forever.
Retry Delay	5	The number of seconds between connection retries.
Service Period	60	The number of seconds before node/group updates are done.
Num Ports	20	Number of ports to search if configured tcpPort is unavailable before porting a failure.

Types

The configuration information that is specific to distributed types. This configuration information is accessed under the Types tab.

Distribution type configuration The information in the following table defines the style of distribution for a given type.

Value	Default	Description
Group Broadcast	Disabled	If enabled, broadcasts of create, write, and deletes are done for all instances of a type.

Value	Default	Description
		When doing the broadcast, all nodes participating in the groups defined for the type will receive the broadcast.
Directed Creation	Disabled	If enabled, all object creates will be directed to the location defined by the location name.
None	Enabled	Default value. If this is enabled, this is a non-distributed object.
Group Broadcast	None	A list of group names associated with this type. Only valid if broadcast is enabled. Currently only one group name is supported.
Location	None	A string containing the node name used for all object creates. Only valid if Directed Create is enabled.

Asynchronous operations The information in the following table defines how asynchronous operations are handled for a given type.

Value	Default	Description
Async. Create	false	If enabled, all creates are sent asynchronously. Only valid if broadcast is enabled.
Async. Delete	false	If enabled, all destroys are sent asynchronously.
Async. Write	false	If enabled, all writes are sent asynchronously.

Caching configuration The information in the following table defines the caching policy for a given type.

Value	Default	Description
Cache Type	always	Cache policy of distribution objects. Valid values are always, never, once, timestamp.
Cache Time	0	Refresh time in seconds. Only valid if cacheType is timestamp.

Flushing configuration The information in the following table defines the flushing criteria for a given type.

Value	Default	Description
Flush Type	None	Flush type of object. Valid values are None, Data and All.

Static discovery configuration

When configuring static discovery, each remote location name that must be contacted from the local node, has the following configuration information. This information is accessed under the Discovery tab in the Distribution Administration User interface.

Value	Default	Description
Location code	None	The location code of the remote node
TCP Port	None	The TCP listener port for the remote node
Host Name	None	The host name for the remote node.

Transactions

This is the configuration information for distributed transactions. It is accessed under the Transactions tab in the Distribution Adminstration User Interface.

Value	Default	Description
Transaction Timeout	60/300	The amount of time to wait, in seconds, for a distributed transaction to establish a transaction lock on an object. If this period expires, a deadlock is thrown, and the global transaction is replayed. The default transaction timeout is increased from 60 to 300 seconds for development builds of the distribution engine.
Transaction Hash Size	101	Hash table size for active global transactions.

Examples

Singleton

This demonstration shows how to implement a global singleton. It uses the following features:

- directed create
- cache never access using create singleton (which refreshes the object if it already exists)
- remote data access using readonly attributes and cache never
- remote data manipulation using operation "setter"

Keys

This demonstration shows how to use group broadcast factories and key lookups to access objects across nodes. It uses the following features:

- group broadcast caching using synchronous writes
- remote operation invocation
- factory lookups and creates using keys
- cache never object access using object references

Exchange

An example of a distributed commodity exchange. It uses group broadcast caching to broadcast price exchanges on commodities (in this case precious metals). It also uses relationships and caching to pull price histories (potentially a large amount of data) across nodes. It uses the following features:

- group broadcast caching using synchronous writes.
- group broadcast caching using asynchronous writes.
- cache never object access using relationship navigation from group broadcast to passively cached objects.
- cache always object access using key lookup of group broadcast cached objects.
- cache always object access using extent iteration of group broadcast cached objects.

Index